

Using Higher-Order Functional Programming to do Register Allocation on a Functional Language

TOMMY HØJFELD OLESEN

hojfeld@diku.dk

(contact person)

MARTIN KOCH

myth@diku.dk

Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø
DENMARK

Abstract. We show how to translate a call-by-value functional language to a RISC architecture in a uniform way that encompasses register allocation and spill code placement, avoids unnecessary copy instructions, provides short-circuit translation of Boolean expressions, and can make use of inter-procedural information. The translation is directed by the source language structure. It uses higher-order functional programming extensively. Preliminary measurements suggest that this method can compete with *graph colouring*, the framework in which most contemporary register allocators are cast. The translation is implemented in the *ML Kit*, a region-inference-based SML compiler. On average, our back end compiles our benchmarks to code that runs in 0.57 of the time of the code generated by SML/NJ version 0.93.

Using Higher-Order Functional Programming to do Register Allocation on a Functional Language

July 9, 1997

Abstract. We show how to translate a call-by-value functional language to a RISC architecture in a uniform way that encompasses register allocation and spill code placement, avoids unnecessary copy instructions, provides short-circuit translation of Boolean expressions, and can make use of inter-procedural information. The translation is directed by the source language structure. It uses higher-order functional programming extensively. Preliminary measurements suggest that this method can compete with *graph colouring*, the framework in which most contemporary register allocators are cast.

The translation is implemented in the *ML Kit*, a region-inference-based SML compiler. On average, our back end compiles our benchmarks to code that runs in 0.57 of the time of the code generated by SML/NJ version 0.93.

1 Introduction

When translating a functional language to intermediate code, many compilers introduce inefficiencies that are supposed to be eliminated by later phases. For instance, they generate temporary variables ad libitum, relying on the register allocator to map these to registers and to eliminate copy instructions. Most contemporary register allocators use *graph colouring* (Chaitin 1982), (Chaitin et al. 1981), which eliminates copy instructions by *coalescing* live ranges in the interference graph (Briggs et al. 1994), (George & Appel 1996).

We propose a “cleaner” way of translating a call-by-value functional language to a RISC architecture that introduces fewer inefficiencies. Instead of using graph colouring, we reexamine earlier methods and allocate registers during code generation. However, in contrast to these earlier methods, we use the source language structure in the register allocation. Whenever a variable is needed during the translation of an expression, we check whether the variable is already in a register; if it is not, a heuristic is used to pick a register for it. The heuristic takes the context of the expression into account. If available, the heuristic will use inter-procedural information about which registers will be changed by a given call, and in which registers a given function wants its parameters. The source language structure is also used for placing spill code.

Our method is inspired by Reynolds’ generation of efficient intermediate code (Reynolds 1995). Having functional values as intermediate representations in the compiler is crucial for this method.

Section 2 gives a flavour of the translation by developing it for the `let`-

construct. Section 3 illustrates how the translation can be extended to give short-circuit translation of Boolean expressions. Section 4 presents measurements. Section 5 discusses related work. Section 6 concludes. An appendix gives an overview of the symbols we use; consult it while reading.

2 Translating an expression to RISC code

2.1 Translating $\text{let } x = e_1 \text{ in } e_2$

The expression

$$\text{let } x = (a+b)+c \text{ in } e_2,$$

could be translated to these RISC (three-address) instructions:

$$\begin{aligned} \phi_1 &:= \phi_a + \phi_b ; \\ \phi_x &:= \phi_1 + \phi_c ; \\ \phi &:= \boxed{\text{code to evaluate } e_2} \end{aligned}$$

where ϕ is the register for the result, ϕ_1 is a temporarily used register, and ϕ_a, ϕ_b, ϕ_c and ϕ_x are the registers allocated to a, b, c and x , respectively. We used a temporary register (ϕ_1) for the sub-expression $a+b$, but not, e.g., for the sub-expressions a or b . The latter two naturally provide a destination register, viz. ϕ_a and ϕ_b , the registers allocated to them. Also the *context* of a sub-expression may naturally provide a destination register. For instance, the context for the argument e_2 in an application $e_1 e_2$ may provide the register that the argument must be passed in as a natural destination register.

A temporary is needed for a given sub-expression iff neither the sub-expression nor its context naturally provides a destination register.

In general, the code for $\text{let } x = e_1 \text{ in } e_2$, using ϕ_x for the register allocated to x , is

$$\phi_x := \boxed{\text{code to evaluate } e_1} ; \phi := \boxed{\text{code to evaluate } e_2}.$$

When translating the **let**-expression, we do not know what the destination register, ϕ , for the entire **let**-expression is. Therefore we do not translate an expression e to code but rather to a function, β , that, when it is applied to the result register, will return code to evaluate e :

$$\beta = \lambda\phi. \phi_x := \boxed{\text{code to evaluate } e_1} ; \phi := \boxed{\text{code to evaluate } e_2}.$$

(The body of a λ -abstraction extends as far to the right as possible.) One may think of β as some code with a hole in it for the destination register. If the sub-expressions e_1 and e_2 are translated to β_1 and β_2 , respectively, the translation of $\text{let } x = e_1 \text{ in } e_2$ can be written

$$\beta = \lambda\phi. \beta_1\phi_x ; \beta_2\phi.$$

Thus the function $\llbracket \cdot \rrbracket_{\text{ra}}$ that translates an expression can be defined for the **let**-construct:

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} &= \text{let } \beta_1 = \llbracket e_1 \rrbracket_{\text{ra}} \\ &\quad \beta_2 = \llbracket e_2 \rrbracket_{\text{ra}} \\ &\quad \text{in } \lambda\phi. \beta_1 \phi_x ; \beta_2 \phi, \end{aligned}$$

where ϕ_x is the register that should contain x .

We also want a sub-expression to tell its context whether it naturally provides a destination register. This is achieved by modifying $\llbracket e \rrbracket_{\text{ra}}$ to furthermore return an optional *natural destination register* $\dot{\phi}$ for e . An absent natural destination register is denoted \bullet . For instance, $\dot{\phi}$ from $\llbracket e_1 + e_2 \rrbracket_{\text{ra}}$ is \bullet .

The natural destination register for **let** $x = e_1$ **in** e_2 is the same as the natural destination register for the sub-expression e_2 :

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} &= \text{let } (\dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \\ &\quad (\dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \\ &\quad \beta = \lambda\phi. \beta_1 \phi_x ; \beta_2 \phi \\ &\quad \text{in } (\dot{\phi}_2, \beta). \end{aligned}$$

If we extend $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$ to decide what ϕ_x above should be, it will be a combined register allocation and code generation for the **let**-construct:

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} &= \text{let } (\dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \\ &\quad (\dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \\ &\quad \boxed{\begin{array}{l} \text{the register allocation part of the translation:} \\ \text{find a register, } \phi_x, \text{ to contain } x \end{array}} \\ &\quad \beta = \lambda\phi. \beta_1 \phi_x ; \beta_2 \phi \\ &\quad \text{in } (\dot{\phi}_2, \beta). \end{aligned}$$

2.2 Register allocation

To keep track of which registers contain which variables, we introduce a *descriptor* δ . The translation function $\llbracket e \rrbracket_{\text{ra}}$ is modified to take and return a δ which describes the contents of the registers at the entry and the exit, respectively, of the code to evaluate e :

$$(\delta_{\text{after}}, \dot{\phi}, \beta) = \llbracket e \rrbracket_{\text{ra}} \delta_{\text{before}}.$$

When a register is needed for a variable x , we have these (not always compatible) four objectives:

1. Preferably choose a register that x is naturally produced in. For instance, in `let $x = e_1$ in e_2` , prefer the natural destination register of e_1 for x .

2. Avoid registers that will be changed while x is live. Consider figure 1. When choosing a register for y , avoid ϕ_1, \dots, ϕ_7 , because they will be changed while y is live, viz. when `f` is called. We say that variable x is *hostile to* ϕ at a given program point iff it is known that ϕ may be changed after that program point while x is live. For instance, y is hostile to ϕ_1, \dots, ϕ_7 at all program points before the call to `f`.

3. Avoid registers that contain live variables. For instance, avoid ϕ_8 for x because δ tells us ϕ_8 contains y which is live.

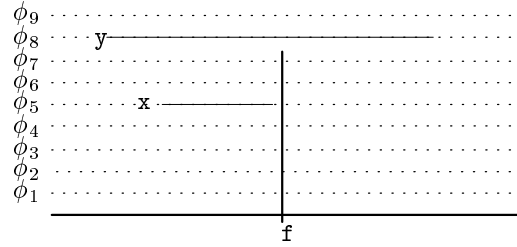


Fig. 1. The bottom line symbolises the code for the expression

```
let y = e1 in
  let x = e2 in (f x)+y.
```

The dotted lines represent registers ϕ_1, \dots, ϕ_9 . The vertical line indicates the point in the code where `f` is called, and which registers will be changed by the call. The horizontal lines x and y indicate the live ranges of variables: x dies before the call to `f`.

4. To minimize the total number of registers changed, prefer a register that is known to be changed anyway. For instance, prefer ϕ_5 to ϕ_9 above, because ϕ_5 will be changed anyway by `f`.

The liveness and hostility information can be collected in one backwards scan. Define ω -*information* to be a map ω from the *live* variables at a given program point to sets of registers that those variables are *hostile to*. Hence, x is live at a program point with ω -information ω iff x is in the domain of ω , and in that case, ωx is the set of registers that x is hostile to.

A preliminary phase annotates ω -information on both sides of each sub-expression of the program. The ω -annotated version of `let $x = e_1$ in e_2` is

$$\omega_3 \left(\text{let } x = \omega_1 e_1 \omega'_1 \text{ in } \omega_2 e_2 \omega'_2 \right) \omega'_3$$

In an *intra*-procedural register allocator, variables that are live across a function will be hostile to the set of caller-saves registers. Inter-procedural information

can be used to make a better approximation (Koch & Olesen 1996). Below, we will omit the ω -annotations and only mention them when needed.

We can define a function $\llbracket \cdot \rrbracket_{\text{def}}$, such that $\llbracket x \rrbracket_{\text{def}} \dot{\phi} \omega \delta$ gives a good choice of register for x , given a preferred register, $\dot{\phi}$, ω -information ω at the current program point, and the current descriptor δ . The heuristic used to define $\llbracket \cdot \rrbracket_{\text{def}}$ is described in section 2.6.

If we arrange that $\llbracket x \rrbracket_{\text{def}} \dot{\phi} \omega \delta$ also returns a δ updated to record that the chosen register now contains x , we can define $\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}}$:

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} \delta_0 &= \text{let } (\delta_1, \dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \delta_0 \\ &\quad (\delta_x, \phi_x) = \llbracket x \rrbracket_{\text{def}} \dot{\phi}_1 \omega_2 \delta_1 \\ &\quad (\delta_2, \dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \delta_x \\ &\quad \beta = \lambda \phi. \beta_1 \phi_x ; \beta_2 \phi \\ &\quad \text{in } (\delta_2, \dot{\phi}_2, \beta). \end{aligned}$$

Here ω_2 is the ω -information before e_2 . Notice how the compile-time δ -flow simulates the run-time control flow. We shall drop the subscripts on δ 's from now on.

2.3 Spilling

When allocating a variable to a register, we may have to throw out some other variable from that register. If the evicted variable is needed later, we must ensure that it is saved in memory, so that it can be reloaded.

Our strategy is very simple: if x is loaded from the stack by the code for e_2 of $\text{let } x = e_1 \text{ in } e_2$, the code for the let -expression must save x on the stack, so that x can be reloaded by the code for e_2 . In other words, if x is loaded by the code for e_2 , the β above should instead be

$$\beta = \lambda \phi. \beta_1 \phi_x ; \text{push } \phi_x ; \beta_2 \phi ; \text{pop}.$$

How do we know whether x is loaded or not? We modify δ to be a pair (c, \mathbf{x}) : The c -component maps registers to a description of their contents. If, e.g., $c\phi = x$, ϕ contains x ; if $c\phi = \emptyset$, ϕ once contained something; if $c\phi = \circ$, ϕ has not been used. We use δ^c to denote the c -component of δ ; etc. The \mathbf{x} -component of δ records which variables must be saved on the stack: If $\llbracket e \rrbracket_{\text{ra}}$ returns δ and $x \in \delta^{\mathbf{x}}$, then x is loaded by the code for e and must be saved on the stack around the code for e .

To keep track, at compile-time, of the position on the stack of variables that are loaded, we use a *stack shape* $\varsigma = (\eta, i)$. This contains a *compile-time stack pointer*, i , (an integer) and an environment η that maps variables to their

stack position (also integers). When we abstract over stack shapes, the β above becomes¹

$$\beta = \lambda\phi. \lambda\varsigma. \beta_1\phi_x\varsigma ; \text{push } \phi_x ; \beta_2\phi(\varsigma^\eta + \{x \mapsto \varsigma^i\}, \varsigma^i + 1) ; \text{pop}.$$

The code for e_1 gets the same stack shape ς as the code for the whole **let**-expression, while the code for e_2 gets the stack shape $(\varsigma^\eta + \{x \mapsto \varsigma^i\}, \varsigma^i + 1)$: The $\varsigma^i + 1$ reflects that one element (x) is pushed around β_2 . The $x \mapsto \varsigma^i$ reflects that x resides at stack offset ς^i .

We use ζ for code abstracted over a stack shape. Thus now, $\beta\phi$ is a ζ , and $\zeta\varsigma$ is actual code. Define $p_{x,\phi}$ to be the function that takes some code ζ and returns code ζ' which preserves ϕ on the stack around ζ :

$$p_{x,\phi}\zeta = \lambda\varsigma. \text{push } \phi ; \zeta(\varsigma^\eta + \{x \mapsto \varsigma^i\}, \varsigma^i + 1) ; \text{pop}.$$

Then β above can be written

$$\beta = \lambda\phi. \lambda\varsigma. \beta_1\phi_x\varsigma ; p_{x,\phi}(\beta_2\phi)\varsigma.$$

We define $\llbracket x \rrbracket_{\text{kill}} \phi_x \delta$ to yield a pair (δ_x, p) , such that, if x is loaded according to δ , p is $p_{x,\phi}$ and δ_x is δ with x removed from the set of loaded variables, i.e., $\delta_x^x = \delta^x \setminus \{x\}$; if x is not loaded according to δ , the p returned by $\llbracket \cdot \rrbracket_{\text{kill}}$ is simply $\lambda\zeta.\zeta$. Using $;$ to glue together ζ 's: $\zeta_1 ; \zeta_2 = \lambda\varsigma. \zeta_1\varsigma ; \zeta_2\varsigma$, we finally have

$$\begin{aligned} \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket_{\text{ra}} \delta &= \text{let } (\delta, \dot{\phi}_1, \beta_1) = \llbracket e_1 \rrbracket_{\text{ra}} \delta \\ &\quad (\delta, \phi_x) = \llbracket x \rrbracket_{\text{def}} \dot{\phi}_1 \omega_2 \delta \\ &\quad (\delta, \dot{\phi}_2, \beta_2) = \llbracket e_2 \rrbracket_{\text{ra}} \delta \\ &\quad (\delta, p) = \llbracket x \rrbracket_{\text{kill}} \phi_x \delta \\ &\quad \beta = \lambda\phi. \beta_1\phi_x ; p(\beta_2\phi) \\ &\text{in } (\delta, \dot{\phi}_2, \beta). \end{aligned}$$

Intuitively, $\llbracket x \rrbracket_{\text{def}}$ appears between $\llbracket e_1 \rrbracket_{\text{ra}}$ and $\llbracket e_2 \rrbracket_{\text{ra}}$ because x “becomes live” between the code for e_1 and that for e_2 .

This way of placing spill code means it is the responsibility of the binder of a variable (e.g., **let** $x = e_1$ **in** e_2) to save that variable (x) if it is loaded. There is no saving of registers around, e.g., function calls: we only record in δ which registers are changed by the function call; if a variable that was thrown out of its register by the call is loaded after the call, this will be recorded in δ , and the variable will be stored by its binder. This *save-at-binding* strategy makes it simple to place the spill code, and it does not have to be done in an ensuing phase. It also means that a variable is saved at most once even when there are several function calls in a row.

¹ $f + \{a \mapsto b\}$ is defined $\lambda x. \text{if } a = x \text{ then } b \text{ else } fx$

2.4 Using a variable

The translation of a use of x in e_2 of **let** $x = e_1$ **in** e_2 is:

$$\begin{aligned} \llbracket x \rrbracket_{\text{ra}} \delta &= \text{if } \exists \phi_x : \delta^c(\phi_x) = x \text{ then } (\delta, \phi_x, \lambda\phi. \lambda\varsigma. \langle \phi := \phi_x \rangle) \\ &\quad \text{else let } (\delta, \phi_x) = \llbracket x \rrbracket_{\text{def}} \bullet \omega\delta \\ &\quad \delta = (\delta^c, \delta^x \cup \{x\}) \\ &\quad \beta = \lambda\phi. \lambda\varsigma. \phi_x := \text{m}[\phi_{\text{sp}} - (\varsigma^i - \varsigma^\eta x)] ; \langle \phi := \phi_x \rangle \\ &\quad \text{in } (\delta, \phi_x, \beta), \end{aligned}$$

where $\langle \phi := \phi' \rangle = \text{if } \phi = \phi' \text{ then } \epsilon \text{ else } \phi := \phi'$, and ϵ is a no-op instruction, and ω is the ω -information before the expression x .

We check whether x is in some register. If it is, the code simply copies that register to the destination register. If x is not in a register, we have to load it. It can be found in the memory cell at offset $\varsigma^i - \varsigma^\eta x$ from the stack pointer. (We assume the stack grows upwards and the register ϕ_{sp} always points to the next free cell.) We must put x in δ^x , such that x 's binder (**let** $x = e_1$ **in** e_2) will save x . Notice we reuse $\llbracket x \rrbracket_{\text{def}}$: the way to choose a register and update δ at a load of x is the same as at the definition of x .

2.5 Functions

A program is translated by translating its functions one at a time. A function is translated by applying first the ω -information phase (section 2.2) and then $\llbracket \cdot \rrbracket_{\text{ra}}$ to it. The resulting β is applied to the register the function should return its result in, and code to return to the caller is appended.

At an application $e_1 e_2$, the natural destination register for e_1 is the register in which the pointer to the closure must be passed; the natural destination register for e_2 is the register in which the argument must be passed; and the natural destination register for $e_1 e_2$ is the register in which the function will return its result. We must remember to record in δ the registers that are changed by the call. By applying the β 's from e_1 and e_2 to specific registers, we force the closure pointer and arguments into the right registers.

For some functions that take a tuple as argument, this register allocation can be extended to pass the elements of the tuple in registers. If we have interprocedural information, we can use different registers at different applications (Koch & Olesen 1996). Otherwise, we can simply use the same convention for all applications.

2.6 The heuristic for choosing registers

This section explains how $\llbracket x \rrbracket_{\text{def}} \bullet \omega\delta$ chooses a register for x .

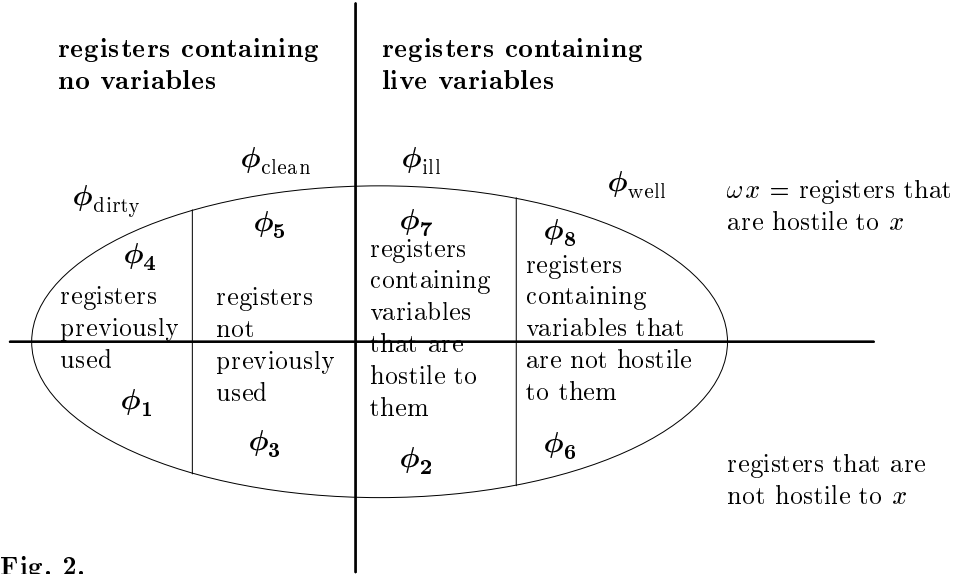


Fig. 2.

I. If $\dot{\phi} = \bullet$: Divide the set of registers into subsets (illustrated in figure 2) that correspond to objectives 2–3 above in the following way:

(i) Aiming at objective 2, divide the registers into those that are hostile to x (i.e., ωx) and those that are not (the horizontal line in the figure).

(ii) Aiming at objective 3, divide the registers according to whether they contain live variables or not (the thick vertical line).

(iii) Aiming at objective 4, divide the registers that do not contain live variables according to whether they will be changed anyway by the current function (ϕ_{dirty}) or not (ϕ_{clean}).

(iv) A variable is *ill-placed* iff it is in a register to which it is hostile, and *well-placed* otherwise. When we are forced to evict a live variable from its register (because all registers contain live variables), it is better to evict an ill-placed variable than a well-placed one. Therefore we divide the registers that contain live variables according to whether they contain ill-placed or well-placed variables (ϕ_{ill} and ϕ_{well} , respectively).

II. In the figure, the pairwise disjoint subsets are numbered in the order we prefer to choose registers from them. For instance, we prefer a register from ϕ_1 to one from ϕ_2 , because the registers in ϕ_1 do not contain live variables, which the registers in ϕ_2 do. The order of the other subsets has been decided with similar considerations.

III. If $\dot{\phi} \neq \bullet$: If $\dot{\phi}$ is not hostile to x , i.e., $\dot{\phi} \notin \omega x$, we choose $\dot{\phi}$, thereby satisfying both objectives 1 and 2. Otherwise, choose a register ϕ as if $\dot{\phi}$ were \bullet , i.e., as described in I–II above. If $\dot{\phi} \notin \omega x$, we elect to satisfy 2 and choose ϕ .

If both ϕ and $\dot{\phi}$ are in ωx , we cannot satisfy **2** and might as well satisfy **1** by choosing $\dot{\phi}$.

This heuristic can be refined in numerous ways, e.g., by taking usage counts into account.

3 Boolean expressions

In our compiler, `if b and i<=j then 3 else 666` is expanded to

`if (if b then i<=j else false) then 3 else 666.`

If we translate Boolean expressions as other expressions, the code for this expression will be as in figure 3(i).

<pre> if $\phi_b = 1$ then ι_1 else ι_4 ; ι_1 : if $\phi_i \leq \phi_j$ then ι_2 else ι_3 ; ι_2 : $\phi_1 := 1$; goto ι_5 ; ι_3 : $\phi_1 := 0$; goto ι_5 ; ι_4 : $\phi_1 := 0$; ι_5 : if $\phi_1 = 1$ then ι_6 else ι_7 ; ι_6 : $\phi := 3$; goto ι_8 ; ι_7 : $\phi := 666$; ι_8 : ϵ </pre>	<pre> if $\phi_b = 1$ then ι_1 else ι_7 ; ι_1 : if $\phi_i \leq \phi_j$ then ι_6 else ι_7 ; goto ι_8 ; ι_6 : $\phi := 3$; goto ι_8 ; ι_7 : $\phi := 666$; ι_8 : ϵ </pre>
(i) naive translation	(ii) “right” translation

Fig. 3. True is represented as 1, false as 0. ι 's are labels. The instruction `if χ then ι else $\bar{\iota}$` jumps to ι if the condition χ is true and to $\bar{\iota}$ otherwise.

We want to avoid the unnecessary manipulation of run-time representations of Boolean values where possible, by translating the expression to short-circuit code as in figure 3(ii). The central idea is not to translate a Boolean expression into code β that accepts a *register*, but rather into a *selector* σ that accepts a *pair of labels* $(\iota, \bar{\iota})$ and returns code that evaluates the Boolean expression and jumps to ι if the result is true and to $\bar{\iota}$ if it is false.

We change $\llbracket \cdot \rrbracket_{\text{ra}}$ to translate a Boolean expression into a σ , while expressions of other types are still translated into a β .

Compare the code for $e_1 + e_2$ (a β) to that for $e_1 \leq e_2$ (a σ) (β_i is the code for e_i):

$$\begin{aligned}
\beta &= \lambda\phi. \lambda\varsigma. \beta_1\phi_1\varsigma ; \beta_2\phi_2\varsigma ; \phi := \phi_1 + \phi_2 \\
\sigma &= \lambda(\iota, \bar{\iota}). \lambda\varsigma. \beta_1\phi_1\varsigma ; \beta_2\phi_2\varsigma ; \text{if } \phi_1 \leq \phi_2 \text{ then } \iota \text{ else } \bar{\iota}.
\end{aligned}$$

Factor the differences out by defining $\llbracket \cdot \rrbracket_o$ by

$$\begin{aligned}\llbracket + \rrbracket_o \phi_1 \phi_2 \phi &= \phi := \phi_1 + \phi_2 \\ \llbracket <= \rrbracket_o \phi_1 \phi_2 (\iota, \bar{\iota}) &= \text{if } \phi_1 \leq \phi_2 \text{ then } \iota \text{ else } \bar{\iota}.\end{aligned}$$

Then β and σ are:

$$\begin{aligned}\beta &= \lambda \phi. \lambda \varsigma. \beta_1 \phi_1 \varsigma ; \beta_2 \phi_2 \varsigma ; \llbracket + \rrbracket_o \phi_1 \phi_2 \phi \\ \sigma &= \lambda (\iota, \bar{\iota}). \lambda \varsigma. \beta_1 \phi_1 \varsigma ; \beta_2 \phi_2 \varsigma ; \llbracket <= \rrbracket_o \phi_1 \phi_2 (\iota, \bar{\iota}).\end{aligned}$$

If we use ξ for something that can be either a register ϕ or a pair $(\iota, \bar{\iota})$ of labels, and τ for something that can be either a β or a σ , the code for $e_1 o e_2$ where o is a binary operator can be written generally:

$$\tau = \lambda \xi. \lambda \varsigma. \beta_1 \phi_1 \varsigma ; \beta_2 \phi_2 \varsigma ; \llbracket o \rrbracket_o \phi_1 \phi_2 \xi.$$

Thus, the implementation of $\llbracket e_1 o e_2 \rrbracket_{\text{ra}}$ is the same for Boolean and non-Boolean expressions.

The σ 's for **true** and **false** jump to the true and false label, respectively:

$$\begin{aligned}\llbracket \text{true} \rrbracket_{\text{ra}} \delta &= (\delta, \bullet, \lambda (\iota, \bar{\iota}). \lambda \varsigma. \text{goto } \iota) \\ \llbracket \text{false} \rrbracket_{\text{ra}} \delta &= (\delta, \bullet, \lambda (\iota, \bar{\iota}). \lambda \varsigma. \text{goto } \bar{\iota}).\end{aligned}$$

The σ for **not** e_1 is obtained by swapping the labels of the σ for e_1 :

$$\llbracket \text{not } e_1 \rrbracket_{\text{ra}} \delta = \text{let } (\delta, \dot{\phi}_1, \sigma_1) = \llbracket e_1 \rrbracket_{\text{ra}} \delta \text{ in } (\delta, \dot{\phi}_1, \lambda (\iota, \bar{\iota}). \sigma_1(\bar{\iota}, \iota)).$$

To translate **if** e_0 **then** e_1 **else** e_2 , translate e_0 to a σ . Apply this to labels ι and $\bar{\iota}$ that label the code for e_1 and e_2 :

$$\begin{aligned}\llbracket \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rrbracket_{\text{ra}} \delta &= \\ \text{let } (\delta_0, \dot{\phi}_0, \sigma) &= \llbracket e_0 \rrbracket_{\text{ra}} \delta \\ (\delta_1, \dot{\phi}_1, \tau_1) &= \llbracket e_1 \rrbracket_{\text{ra}} \delta_0 \\ (\delta_2, \dot{\phi}_2, \tau_2) &= \llbracket e_2 \rrbracket_{\text{ra}} \delta_0 \\ \delta &= \delta_1 \sqcap \delta_2 \\ \tau &= \lambda \xi. \lambda \varsigma. \sigma(\iota, \bar{\iota}) \varsigma ; \iota : \tau_1 \xi \varsigma ; \text{goto } \check{\iota} ; \\ &\quad \bar{\iota} : \tau_2 \xi \varsigma ; \quad \check{\iota} : \epsilon \\ &\text{in } (\delta, \bullet, \tau),\end{aligned}$$

where ι , $\bar{\iota}$, and $\check{\iota}$ are fresh labels. Notice the compile-time δ -flow reflects the run-time control flow. $\delta_1 \sqcap \delta_2$ yields a δ that is a safe combination of δ_1 and δ_2 .

To get full short-circuit translation (e.g., of **and**), it is an important point that also the branches of the **if** may translate to σ 's.

A Boolean expression may translate to a β but occur in a context that needs a σ (e.g. as **f 7** of **if f 7 then e_1 else e_2**). *mk-selector* converts a β to a σ that checks what truth value the β computes and jumps to the corresponding label. Assume $\dot{\phi}$ and δ are the natural destination register and descriptor that correspond to β , and ω is the relevant ω -information:

$$\begin{aligned} mk-selector(\delta, \dot{\phi}, \beta)\omega &= \text{let } (\delta, \phi) = tmp \dot{\phi} \omega \delta \\ &\quad \sigma = \lambda(\iota, \bar{\iota}). \lambda\varsigma. \beta\phi\varsigma ; \text{if } \phi = 1 \text{ then } \iota \text{ else } \bar{\iota} \\ &\quad \text{in } (\delta, \bullet, \sigma). \end{aligned}$$

$tmp \dot{\phi} \omega \delta$ is much like $\llbracket \cdot \rrbracket_{\text{def}}$; it chooses a register (preferably $\dot{\phi}$) to hold a value temporarily, and updates δ accordingly.

The converse situation may occur; e.g., in **let $b=a<=c$ in e** we want the result of $a<=c$ as a value in a register. *mk-beta* converts a σ to a β :

$$\begin{aligned} mk-beta(\delta, \dot{\phi}, \sigma) &= \\ &\quad \text{let } \beta = \lambda\phi. \lambda\varsigma. \sigma(\iota, \bar{\iota})\varsigma ; \iota : \phi := 1 ; \text{goto } \check{\iota} ; \bar{\iota} : \phi := 0 ; \check{\iota} : \epsilon \\ &\quad \text{in } (\delta, \dot{\phi}, \beta). \end{aligned}$$

Now, the example from above will generate the code in figure 3(ii) (assuming jumps to jumps are eliminated, which is best done when the graph of RISC-like instructions we generate is flattened to linear code).

4 Assessment

We have implemented a PA-RISC back end (OK) that incorporates the translation discussed here as part of an *inter-procedural register allocator*. It is implemented in the *ML Kit*, an SML compiler based on *region inference*, which infers, at compile-time, when memory can be allocated and deallocated and thus makes garbage collection unnecessary (Birkedal et al. 1996). The translation does not rely on being in a region-inference-based compiler; it could work with garbage collection as well.

We compare with KAM, another back end for the ML Kit, (Birkedal 1994), (Elsman & Hallenberg 1995). The main differences between the two are:

- 1° KAM uses graph colouring.
- 2° Around each function call, KAM saves all live variables, whereas OK's save-at-binding strategy means that a variable is saved at most once.
- 3° Boolean expressions: for **if $a<=b$ then e_1 else e_2** both back ends generate short-circuiting code, while for **andalso** only OK does.

- 4° KAM fetches a free variable from the closure every time it is used.
- 5° Exceptions are translated differently.
- 6° KAM's register allocator works on basic blocks; OK works on functions.
- 7° OK uses inter-procedural information.
- 8° OK lets more registers participate in the register allocation.
- 9° OK allows functions to pass several arguments in registers.
- 10° OK duplicates code to avoid jumps.
- 11° OK does instruction scheduling.

On average, we compile the benchmarks below to code that runs in 0.75 of the time of the code generated by KAM, and in 0.57 of the time of the code generated by SML/NJ (Appel 1992) version 0.93; see (Koch & Olesen 1996).

Comparing the time spent in the back ends of KAM and OK, KAM is approximately twice as fast as OK. This seems ok: KAM is well-tuned; OK is still a prototype and we have invested no effort in reducing compile-time.

k kb	Knuth-Bendix completion, improved for region inference by Mads Tofte.
life	life, using lists, improved for region inference.
appel, bappel	function application in a row, simple arithmetic.
ip, plusdyb	3 and 7 functions deep call graph in a loop, respectively.
ack, tak	multiple-argument functions (Ackermann and Takeuchi)
fib	Fibonacci
bul	should benefit from our short-circuit translation of Boolean expressions.
fri	a function that uses its free variables many times.
handle	introduce handlers often, raise exceptions only exceptionally.
raise	raise a lot of exceptions.
reynolds ryenolds	designed to exhibit good, respectively, bad, behaviour with region inference (Birkedal et al. 1996)
church	arithmetic using Church numerals, many function applications and fetches of free variables from a closure.
foldr	build and fold a big constant list.
msort, qsort	sorting
iter	compute $(f \circ \dots \circ f)a$ for different f 's and a 's.

	(i) normal <i>choose</i>		(ii) $\omega = \emptyset$	(iii) same always	(iv) OK, intra- procedural version	(v) KAM
kfb	18.52 s	1.00	1.13	1.41	1.20	1.64
life	17.40 s	1.00	1.73	2.38	1.34	1.78
appel	10.38 s	1.00	1.29	1.70	1.19	1.20
bappel	18.64 s	1.00	1.25	1.62	1.15	1.23
ip	8.70 s	1.00	1.56	3.02	1.19	1.58
plusdyb	14.41 s	1.00	1.20	1.47	1.22	1.19
ack	10.00 s	1.00	1.00	1.90	1.16	1.29
fib	43.31 s	1.00	0.94	1.26	0.96	1.00
tak	20.32 s	1.00	1.05	1.49	1.61	1.49
bul	11.98 s	1.00	1.09	1.36	1.06	1.29
fri	7.85 s	1.00	1.32	2.88	1.06	1.52
handle	27.93 s	1.00	1.09	1.37	1.10	1.18
raise	19.29 s	1.00	1.12	1.10	1.01	1.37
ryenolds	14.86 s	1.00	1.03	1.21	0.99	1.05
reynolds	11.26 s	1.00	1.02	1.08	0.99	0.87
church	34.00 s	1.00	1.05	1.28	1.03	1.11
foldr	24.22 s	1.00	1.23	1.54	1.11	1.30
msort	7.26 s	1.00	1.31	1.57	1.11	1.60
qsort	20.02 s	1.00	1.37	1.70	1.04	1.38
iter	14.76 s	1.00	1.36	1.80	2.01	1.78
<i>geom. mean</i>		1.00	1.19	1.59	1.16	1.32

The tables above try to assess the heuristic described in section 2.6: Column (i) gives the run-time of the benchmarks compiled with OK. All other columns are normalised to this. In (ii), the heuristic is blinded by assuming ω is always \emptyset . In (iii), the heuristic chooses the same register whenever possible. This (roughly) gives a lower bound on how bad a heuristic can do.

It seems the heuristic is quite good: (i) is much better than (ii), which itself is not unreasonably bad as it is much better than (iii), the “lower bound”.

Attempting to compare the register allocation described here with graph colouring, we have tried making an *intra*-procedural version of OK, i.e., giving it the same conditions as KAM: no inter-procedural information, i.e., uniform calling conventions, total caller-saves convention (7°), restricting the set of available registers to approximately what KAM uses (8°), only pass one argument in a register (9°), no code duplication (10°), and no instruction scheduling (11°). Thus, the two back ends still differ on 1° through 6°, although we would prefer that they only differed on 1° (and maybe 2°).

With these inhibitions on OK, the mean run-time increases to 1.16 (iv), which is still smaller than KAM’s 1.32 (v). This suggests that the translation

presented here can compete with graph colouring, but this is not conclusive as the two register allocators are different in other aspects (notably 6°).

We have two reservations concerning the experiments: many benchmarks are toy programs, and the timing of the benchmarks may be inaccurate; the number of decimals is not an indication of the accuracy of the measurements.²

5 Comparison with other work

The way we assign variables to registers means that our register allocator allows a variable to reside in different registers and in memory in different parts of the program; it does not utilise this in any systematic way, however. In the basic graph-colouring framework, a variable is either allocated to a register for all of its live range or not at all; it cannot be put in different places in different parts of its live range. (Extending basic graph colouring with *live range splitting* may, however, circumvent some of these problems (Chow & Hennessy 1990).)

Our translation avoids building and maintaining an interference graph, which makes graph-colouring register allocation expensive (Gupta et al. 1994).

Our translation gives a natural way of placing spill code using the source language structure. This is not the case for graph colouring, because the use of an interference graph separates the register allocation problem from the program from which it originated. Using program structure for register allocation has recently been investigated in (Callahan & Koblenz 1991), (Thorup 1995) and (Kannan & Proebsting 1995).

One can envision situations where our algorithm with its somewhat local choice does worse than graph colouring with the more global perspective the interference graph gives. On the other hand, graph colouring also relies on heuristics, and possibly, our method does as well in practice as many heuristics for colouring graphs. Furthermore, the things that the graph-colouring framework for register allocation is not good at addressing (e.g., spill code placement, allowing the same variable to be in different places) may well influence the quality of the register allocation more than the things it is good at addressing.

Choosing registers as code is emitted is not a new technique (Waite 1974); it predates graph colouring. Pre-graph-colouring methods for register allocation do not, however, exploit the program structure, and their choice of register is not based on concepts like our ω -information or natural destination register.

(Hsu et al. 1989) presents a register allocation algorithm that also chooses

²The experiments were run on an unloaded HP 9000/C100 with 256MB RAM. Timing results is the minimum sum of the “user” and “system” time as measured by Unix **time** after running the benchmark thrice.

registers as code is emitted based on the distances to the next uses of variables. They only discuss basic-block-level register allocation and argue that in that setting, their algorithm does better than graph colouring. They do not exploit the program structure.

A common way of bridging the gap between a call-by-value functional source language and the imperative target language is to first transform the program into *continuation-passing style* (Kranz et al. 1986), (Appel 1992). This method relies on ensuing phases to clean up the generated code.

Our use of functional values in the translation is inspired by (Reynolds 1995) (which is not about register allocation). For instance, the way we avoid copy instructions is related to the way unnecessary temporary variables are avoided.

Short-circuit translation of Boolean expressions is not new (Aho et al. 1986). In contrast to many short-circuiting translations, we generate short-circuit code for **andalso** and **orelse** without treating these constructs explicitly. The crucial thing to do this is to generate short-circuit code for **if** (**if** ...) **then** e_1 **else** e_2 . By doing this, we get short-circuit code in more situations (e.g., we get short-circuit code for (**case** **ses** of **Some** $x \Rightarrow x$ | **None** \Rightarrow **false**) **andalso** e), and the compiler is simpler with fewer source language constructs. This is not new either: (Brooks et al. 1982) achieves the same. However, our short-circuiting translation comes naturally from a small change in the general translation method, while (Brooks et al. 1982) relies on preceding transformations to transform **if**-expressions into special forms which the compiler can translate to short-circuit code.

6 Conclusions

We have succeeded in extending the approach illustrated here for the **let**-construct to all constructs in our source language: references, exceptions, function application, etc. In some respects, the translation generates code that is more “true to” the functional source code than more naive translations that rely on ensuing phases to clean up the generated code. The algorithm is more complicated, but since its source language is fairly small, this is perhaps not a problem. A good sign is that short-circuit translation of Boolean expressions falls directly into our lap when we abstract the code over a pair of destination labels $(\iota, \bar{\iota})$ instead of over a destination register ϕ . Furthermore, the translation can make good use of inter-procedural information: using it will decrease the run-time of the benchmarks with 7% on average (Koch & Olesen 1996).

Our translation avoids some of the unfelicities with graph colouring. On the other hand, because our translation is so closely coupled to the structure of the program, it may hinder register allocation optimisations that want to change

the order of code. For instance, (Sethi & Ullman 1970) changes the evaluation order to minimise the number of registers needed to evaluate an expression, and (Burger et al. 1995) changes the evaluation order of function arguments to make a good “shuffling” of registers at function applications. But note that the order of evaluation can often not be changed anyway in SML, because many expressions have side effects (e.g., evaluating $x+y$ may raise a `Sum` exception).

Comparisons of object code quality with a graph-colouring register allocator are in our favour, but this is not conclusive as the two register allocators are different in other aspects. Graph colouring has almost completely conquered the world of register allocation and it is a conceptually nice method, but it seems that other methods can compete in terms of efficiency.

ACKNOWLEDGMENTS

We want to thank Finn Schiermer Andersen, Lars Birkedal, Erik Bjørnager Dam, Martin Elsmann, Sasja Frahm, Arne John Glenstrup, Niels Hallenberg, Jo Koch, Torben Mogensen, Kristian Nielsen, and Mads Tofte. This article is based on (Koch & Olesen 1996), on which Mads Tofte was supervisor.

REFERENCES

- Aho, Alfred V., Ravi Sethi & Jeffrey D. Ullman (1986): *Compilers Principles, Techniques and Tools*. Reading, Massachusetts.
- Appel, Andrew W. (1992): *Compiling with Continuations*. Cambridge.
- Birkedal, Lars (1994): *The ML Kit Compiler—Working Note*. Unpublished manuscript.
- Birkedal, Lars, Mads Tofte & Magnus Vejlstrup (1996): From region inference to von Neumann machines via region representation inference. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersburg Beach, Florida. 171–183.
- Briggs, Preston, Keith D. Cooper & Linda Torczon (1994): Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems* **16**(3), 428–455.
- Brooks, Rodney A., Richard P. Gabriel & Guy L. Steele, Jr. (1982): An optimizing compiler for lexically scoped Lisp. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (= *SIGPLAN Notices* **17**(6)). Boston, Massachusetts. 261–275.
- Burger, Robert, Oscar Waddell & R. Kent Dybvig (1995): Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* (= *SIGPLAN Notices* **30**(6)). La Jolla, California. 130–138.
- Callahan, David & Brian Koblenz (1991): Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (= *SIGPLAN Notices* **26**(6)). 192–203.

- Chaitin, Gregory J. (1982): Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction (= SIGPLAN Notices 17(6))*. Boston, Massachusetts. 98–105.
- Chaitin, Gregory J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins & Peter W. Markstein (1981): Register allocation via coloring. *Computer Languages 6*, 47–57.
- Chow, Fred C. & John L. Hennessy (1990): The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems 12(4)*, 501–536.
- Elsman, Martin & Niels Hallenberg (1995): An optimizing backend for the ML Kit using a stack of regions. Student Project 95-7-8. Department of Computer Science, University of Copenhagen.
- George, Lal & Andrew W. Appel (1996): Iterated register coalescing. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersburg Beach, Florida. 208–218.
- Gupta, Rajiv, Mary Lou Soffa & Denise Ombres (1994): Efficient register allocation via coloring using clique separators. *ACM Transactions on Programming Languages and Systems 16(3)*, 370–386.
- Hsu, Wei-Chung, Charles N. Fischer & James R. Goodman (1989): On the minimization of loads/stores in local register allocation. *IEEE Transactions on Software Engineering 15(10)*, 1252–1260.
- Kannan, Sampath & Todd Proebsting (1995): Register allocation in structured programs. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. 360–368.
- Koch, Martin & Tommy Højfeldt Olesen (1996): *Compiling a Higher-Order Call-by-Value Functional Programming Language to a RISC Using a Stack of Regions*. Master's thesis, Department of Computer Science, University of Copenhagen.
<http://www.diku.dk/students/hojfeldt/masters.html>
- Kranz, David, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin & Norman Adams (1986): ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction (= SIGPLAN Notices 21(7))*. Palo Alto, California. 219–233.
- Reynolds, John C. (1995): Using functor categories to generate intermediate code. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California. 25–36.
- Sethi, Ravi & J. D. Ullman (1970): The generation of optimal code for arithmetic expressions. *Journal of the ACM 17(4)*, 715–728.
- Thorup, Mikkel (1995): *Structured Programs have Small Tree-Width and Good Register Allocation*. Technical Report 95/18. Department of Computer Science, University of Copenhagen.
- Waite, W. M. (1974): Code generation. In F. Bauer & J. Eickel (eds.): *Compiler Construction—An Advanced Course (= Lecture Notes in Computer Science 21)*. Berlin. Chap. 3E, 302–332.

Symbols used

Here is an overview of the symbols used. Be aware that we refine the definition of some symbols during the exposition; e.g., at first β 's are in $\Phi \rightarrow K$, later they are in $\Phi \rightarrow Z$.

It is implicit in a symbol which set it ranges over; e.g., x is always in X . We use $\mathcal{P}A$ for the set of subsets of A ; and ς^η denotes the η -component of ς ; etc.

$e \in E ::= \text{let } X = E \text{ in } E \mid X \mid E E \mid \lambda X. E$	expression (the source language)
$\mid \text{if } E \text{ then } E \text{ else } E \mid \text{true} \mid \text{not } E$	
$\mid E \leq E \mid E + E \mid I \mid \dots$	
$x \in X$	variable
$\mathbf{x} \in \mathcal{P}X$	set of variables
$\kappa \in K ::= \Phi := m[\Phi - I] \mid \Phi := \Phi$	RISC instruction (the target language)
$\mid L : K \mid \text{goto } L \mid \text{if } \Phi = I \text{ then } L \text{ else } L$	
$\mid \text{push } \Phi \mid \text{pop} \mid K ; K \mid \epsilon \mid \dots$	
$\phi \in \Phi$	register
$\dot{\phi} \in \dot{\Phi} = \Phi \cup \{\bullet\}$	natural destination register
$\iota \in L$	label
$\beta \in B = \Phi \rightarrow Z$	code (abstracted over destination register)
$\zeta \in Z = S \rightarrow K$	code (abstracted over stack shape)
$p \in P = Z \rightarrow Z$	preserver
$(\eta, i) = \varsigma \in S = (X \rightarrow I) \times I$	stack shape
$\sigma \in \Sigma = (L \times L) \rightarrow Z$	selector
$\tau \in T ::= B \mid \Sigma$	code or selector
$\xi \in \Xi ::= \Phi \mid L \times L$	destination (register or labels)
$i \in I$	integer
$(c, \mathbf{x}) = \delta \in \Delta = (\Phi \rightarrow D) \times \mathcal{P}X$	descriptor
$d \in D ::= \otimes \mid \circ \mid X \mid \dots$	description
$\omega \in \Omega = X \rightarrow \mathcal{P}\Phi$	ω -information

$$\llbracket \cdot \rrbracket_{\text{ra}} \in E \rightarrow \Delta \rightarrow (\Delta \times \dot{\Phi} \times B)$$

$$\llbracket \cdot \rrbracket_{\text{def}} \in X \rightarrow \dot{\Phi} \rightarrow \Omega \rightarrow \Delta \rightarrow (\Delta \times \Phi)$$

$$\llbracket \cdot \rrbracket_{\text{kill}} \in X \rightarrow \Phi \rightarrow \Delta \rightarrow (\Delta \times P)$$